Name: Muhammad Taha Navaid
Professor: Jawahar Panchal
Class: CS 429 - Information Retrieval
Date: 4/30/2022

Abstract

In this python-based project, a program was developed to take user queries as inputs and output a set of documents that match closely with that query.

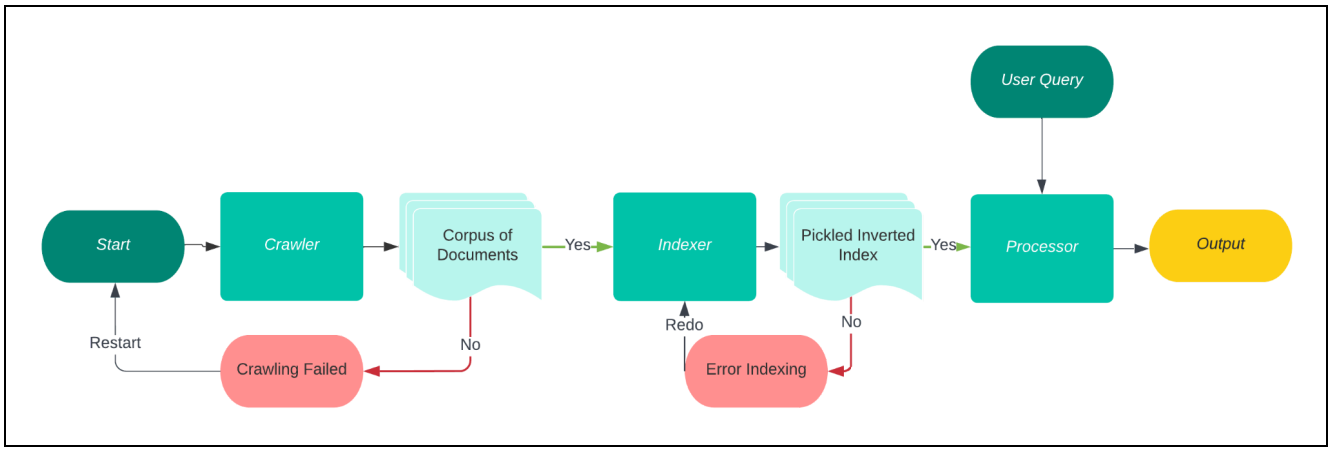
The purpose of this project was to use the knowledge gained during the course and combine it with personal artistry to create a mini search engine that exhibits basic query search functionality, demonstrates the ability to construct and use inverted indexes with tf-idf scorings, provides exposure to useful python libraries (i.e. Flask, Scikit Learn, Scrapy, etc.), and create solid foundations for understanding the entire system: from web crawling to processing, and everything in between.

Future steps could involve enhancing the usage capabilities of the system by allowing different types of files to be scraped from the web, readjusting to the model for better performance and faster processing, improving the accuracy of the query search results, etc.

Overview

The project can be broken down into 3 significant parts:

1) A Scrapy based Crawler for downloading web documents in HTML format - content crawling:
    - This part should crawl a web page, given a URL, and can take information from a user-specified number of maximum pages, and to a maximum depth
2) A Scikit-Learn based Indexer for constructing an inverted index in pickle format - search indexing:
    - An inverted index, using tf-idf scores and cosine similarity, should be created for the documents crawled from part 1, and stored as a pickle file for later retrieval.
3) A Flask based Processor for handling free text queries in JSON format - query processing:
    - This is where python Flask is utilized for retrieving the pickle[d] inverted index and processing user queries.



Design

The three main architectural components of the system are directly or indirectly linked to each other. The indexer needs the correct corpus of documents with the proper file format to continue indexing and the processor needs the data to be properly structured and has an appropriate inverted index with tf-idf and cosine similarity values to output useful results.

One part of the system going wrong can affect the other parts of the program and its performance.

Architecture

A main, master folder was created (named 'Project') to store all of the files that will be used to create the system.

A python file, named '**Scrapy_Crawler.py**', was created to implement the starting code for the program. Scrapy was installed and its components were placed in the 'Project' folder.

Scrapy crawlers come in 4 templates: 1) General purpose, 2) Crawl spider, 3) CSV parser, and 4) XML parser. For this system, the crawl spider is used as it best fits the functional requirements.

In the main 'Project' folder, a subfolder named 'downFiles' was created. This will be used to store the infrastructure of the scrapy spider, and contain the crawler python script ('Scrapy_Crawler.py') as well.

The 'downFiles' subfolder includes the 5 main scripts needed to tweak and run the scrapy crawler. These scripts are called: '**_init_.py**', '**items.py**', '**middlewares.py**', '**pipelines.py**', and '**settings.py**'.

The following changes were made to the aforementioned scripts before running the main scrapy spider:

1) In the '**settings.py**' script, the following code was added to allow the spider to have certain functionalities:

a)
```
ITEM_PIPELINES = {
    'scrapy.pipelines.files.FilesPipeline': 1,
}
```

   i) The controls the downloading of files

b)
```
FILES_STORE = r"D:\College\College Courses\CS 429\Project"
```

   i) This specifies the destination where the downloaded files will be stored.

2) In the '**items.py**' file, a function is added to define the fields for the item:

a)
```
class DownfilesItem(scrapy.Item):
    # define the fields for your item here like:
    file_urls = scrapy.Field()
    original_file_name = scrapy.Field()
    files = scrapy.Field
```

After making the changes, the scrapy python script is coded and named '**Scrapy_Crawler.py**'.

Deconstructing the code:

1) First, all the libraries and dependencies are imported into the script
2) A "Crawler" class is created, which holds the name of the spider as 'Scrapy_Crawler'

```
class Crawler(CrawlSpider):
    name = 'Scrapy_Crawler'
    allowed_domains = ['www.nirsoft.net'] # Here put desired domain restriction
    start_urls = ['http://www.nirsoft.net/'] # Here put desired url

    Max_Pages = 1000 # Here change how many maximum pages you would like.
    Max_Depth = 1 # Here change desired maximum depth


    count = 0 # The count starts at zero.

    custom_settings = {
        'DEPTH_LIMIT': str(Max_Depth),
    }

    rules = (
        Rule(LinkExtractor(allow=''),
         callback='parse_item', follow = True),
    )
```

a) Naming the spider is important because that allows an external shell/script to run the spider.

i)
```
name = 'Scrapy_Crawler'
```

b) Two variables **Max_Depth** and **Max_Pages** are created to allow the user for setting the limits on those variables. These parameters are used to control how much the spider crawls from the given initial URL.

i)
```
Max_Pages = 1000 # Here change how many maximum pages you would like.
Max_Depth = 1 # Here change desired maximum depth
```

c) The **count** variable controls the number of times that the iteration will occur which is directly linked to the earlier **Max_Depth** variable, and stops the spider from crawling once it has reached that user-specified depth.

i)
```
count = 0 # The count starts at zero.
```

d) The **rule** dictionary contains the LinkExtractor library, which gets the links from the crawled websites.

i)
```
rules = (
    Rule(LinkExtractor(allow=''),
     callback='parse_item', follow = True),
)
```

3) A "parse_item" function is defined within the "Crawler" class to parse the crawled pages, as well as stop the spider from crawling further once the **count** limit has been reached. This is done by adding a conditional if statement to the function, and setting the condition to if (self. **count** >= self. **Max_Pages**), stop the crawling.

i)

```
def parse_item(self, response):
        # Return if more than N
    if self.count >= self.Max_Pages:
        raise CloseSpider(f"Scraped {self.N} items. Eject!")
    # Increment to count by one:
    self.count += 1

    file_url = response.css('.downloadline::attr(href)').get()
    file_url = response.urljoin(file_url)
    file_extension = file_url.split('.')[-1]
    if file_extension not in ('html'):
        return
    item = DownfilesItem()
    item['file_urls'] = [file_url]
    item['original_file_name'] = file_url.split('/')[-1]
    yield item
```

A .ipynb file is used to run the spider from the main project directory. This runs the crawler and gets the HTML documents of the crawled pages. The retrieved HTML documents are then stored in a folder called 'full' inside the …/Project directory.

To implement the next operation, another python script named '**Scikit Indexer.py**' is created and the needed libraries are imported. Since the documents are already in a folder, the files are imported with the *codecs* library and are iterated into a list using a simple for loop.

```
# iterate through all files
for file in os.listdir():
    file_path = f'{path}/{file}'
    c.append(str(codecs.open(file_path, 'r').read()))
```

Then, another for loop is used to store the retrieved documents in a variable called **corpus** by removing the unnecessary symbols and extra spaces from the collection. This allows for the data to be more structured and usable.

```
for i in range(len(c)):
    corpus.append(re.sub("[^a-zA-Z0-9]+", " ",c[i]))
```

Next, a variable **clist** is initialized. The **clist** variable is a list of lists, and a nested for loop is used to individually place each word in the document **corpus** into this list.

```
clist = []
for i in range(len(corpus)):
    clist.append([])
    temp = ''
    for j in corpus[i]:
        if j != ' ':
            temp = temp+j
        if j == ' ':
            clist[i].append(temp)
            temp = ''
```

A function is written with the name of **tfIDF_Index()**, which takes in the list of words in each document (in this case **clist**), and creates an inverted index containing the tf-idf scores for each term, relative to the documents that contain them.

```
def tfIDF_Index(a):
    inv_idx = {}
    sym = [',', "'", '.']
    for i in range(len(a)):
        for j in range(len(a[i])):
            if a[i][j] not in sym:
                if a[i][j].lower() not in inv_idx:
                    inv_idx[a[i][j].lower()] = [[i]]
                if a[i][j].lower() in inv_idx:
                    st = a[i][j].lower()
                    if i > inv_idx[st][len(inv_idx[st])-1][0]:
                        inv_idx[st].append([i])
    for i in inv_idx:
        for j in range(len(inv_idx[i])):
            tf = 0
            for k in a[inv_idx[i][j][0]]:
                if i == k.lower():
                    tf += 1
            tf = tf/len(a[inv_idx[i][j][0]])
            inv_idx[i][j].append(tf*math.log(len(a)/len(inv_idx[i])))

    return inv_idx
```

The above function is used to create an inverted index and store it in a variable called **inv_index**.

```
inv_index = tfIDF_Index(clist)
```

Now, the cosine similarity processing has to be added to the script. To assist with this, the *sklearn* library and further dependencies are imported. The **corpus** is given to the *sklearn* function **CountVectorizer()** and stored (after transformation) to variable **sparse_matrix**. After that, the matrix is converted to a data frame (variable **df**) and the output of the first few lines of **df** is:

	00	000	000000	000080	0000ff	008	008000	01	011	012	...	your	youtube	zero	zhtwnet	zip	zipgenius	zipinst	zipinstaller	zone	zones
0	0	0	0	1	0	0	0	0	0	0	...	4	0	0	0	2	0	0	0	0	0
1	1	0	0	1	0	0	0	3	0	0	...	3	0	0	1	11	0	0	0	0	0
2	0	0	0	1	0	0	0	0	0	0	...	4	0	0	0	0	0	0	0	0	0
3	2	0	0	1	0	0	0	3	0	0	...	9	0	0	1	12	0	0	0	0	0
4	0	0	0	1	0	0	0	0	0	0	...	29	0	0	0	5	0	0	0	0	0
5	0	0	0	1	0	0	0	0	0	0	...	26	0	0	0	23	3	2	3	0	0
6	0	0	0	1	0	0	0	0	0	0	...	2	0	0	0	0	0	0	0	0	0
7	0	0	0	1	0	0	0	0	0	0	...	11	0	1	0	3	0	0	0	0	0
8	0	0	0	1	0	0	0	0	0	0	...	2	0	0	0	1	0	0	0	0	0
9	0	0	0	1	0	0	0	0	0	0	...	4	0	0	0	0	0	0	0	0	0
10	0	0	0	1	0	0	0	0	0	0	...	15	0	0	0	0	0	0	0	0	0

This **df** is then used in the *sklearn* **cosine_similarity** function to convert the values, and stored in the variable **dj.**

```
# Cosine Similarity

dj=pd.DataFrame(cosine_similarity(df, dense_output=True))
```

The data frame is as follows:

	0	1	2	3	4	5	6	7	8	9	...	20	21	22	23	24	
0	1.000000	0.860304	0.938529	0.854952	0.893571	0.870838	0.879368	0.878399	0.914151	0.904602	...	0.846642	0.836276	0.878427	0.885740	0.831018	0
1	0.860304	1.000000	0.854430	0.936972	0.780108	0.833273	0.865642	0.801160	0.863974	0.849338	...	0.921848	0.930392	0.799100	0.874757	0.776294	0
2	0.938529	0.854430	1.000000	0.845063	0.831344	0.854330	0.911977	0.858764	0.912783	0.911069	...	0.834786	0.824992	0.841826	0.909575	0.815167	0
3	0.854952	0.936972	0.845063	1.000000	0.800444	0.854099	0.855260	0.792826	0.847973	0.840577	...	0.924631	0.932227	0.806299	0.867482	0.790383	0
4	0.893571	0.780108	0.831344	0.800444	1.000000	0.863288	0.745374	0.873882	0.831545	0.778304	...	0.801388	0.787984	0.928486	0.775131	0.819832	0
5	0.870838	0.833273	0.854330	0.854099	0.863288	1.000000	0.835506	0.841509	0.843358	0.838163	...	0.844664	0.830547	0.838462	0.843366	0.858901	0
6	0.879368	0.865642	0.911977	0.855260	0.745374	0.835506	1.000000	0.789122	0.875738	0.894258	...	0.837669	0.838183	0.761866	0.961748	0.775609	0
7	0.878399	0.801160	0.858764	0.792826	0.873882	0.841509	0.789122	1.000000	0.842321	0.821323	...	0.833148	0.790807	0.849488	0.800080	0.797589	0
8	0.914151	0.863974	0.912783	0.847973	0.831545	0.843358	0.875738	0.842321	1.000000	0.885409	...	0.829720	0.826602	0.877642	0.880470	0.815766	0
9	0.904602	0.849338	0.911069	0.840577	0.778304	0.838163	0.894258	0.821323	0.885409	1.000000	...	0.823062	0.820408	0.783999	0.892588	0.814607	0
10	0.863002	0.850045	0.875723	0.851452	0.728499	0.835187	0.888551	0.771940	0.850663	0.861098	...	0.831704	0.828091	0.756522	0.881272	0.789410	0

Finally, the *pickle* library is imported and used to take the updated inverted index and stored in the 'Project' folder for later usage. *Pickle* is also utilized to convert and store the **clist** variable in the main folder.

```
# User specified directory, set as you wish
filename = "D:/College/College Courses/CS 429/Project/index.pkl"
os.makedirs(os.path.dirname(filename), exist_ok=True)

# This will download the index as a pickle file in the '.../Project' directory
with open('D:/College/College Courses/CS 429/Project/index.pkl', 'wb') as f:
    pickle.dump(inv_index, f)

with open('D:/College/College Courses/CS 429/Project/corpus.pkl', 'wb') as ft:
    pickle.dump(clist, ft)
```

At last, the python script '**Flask Processor.py**' is created to process the user queries by retrieving the indexed data.

```
Enter query: programming

Set top K limit (integer number): 10
 * Serving Flask app "Flask Processor" (lazy loading)
 * Environment: production
   WARNING: This is a development server. Do not use it in a production deployment.
   Use a production WSGI server instead.
 * Debug mode: off
 * Running on http://127.0.0.1:5000/ (Press CTRL+C to quit)
```

Top K Ranked Documents : {0: '0', 1: '19', 2: '28'}

Operation

Order of Operations:

1) First, download the zip file and place it into a user-specified folder.
2) Then, unzip the file and extract all of the components into the folder.
3) Take the 'full' file out and place it somewhere else. Do NOT delete it because if the spider fails to work correctly, there is always a set of HTML documents already parsed that can be used for the following procedures
4) After that, open the '**Scrapy_Crawler.py**' and set the desired initial URL.

a) In this script, the user will have to set the URL two times, with the **allowed_domains** variable only taking the 'www.' version, while the **start_urls** takes the extended 'https://' version of the same URL.
b) In the script, the user can adjust the limit of **Max_Pages** and **Max_Depth** as desired

5) Open the spider_run.ipynb file, set the directory to the …/'User Folder'/downFiles, and run the spider using the third command line.
6) At the end of the run, the spider should close and download a set of documents in a folder called 'full' at the directory location which contains the HTML documents.
7) Next, open the '**Scikit Indexer.py**' script and change where the pickle should download the inverted index to the user-specified folder directory.
8) Then, run the script and it should do the necessary operations on the HTML file data, and store the index as a pickle file in the folder.
9) Finally, open the '**Flask Processor.py**' script and run it. It will ask the user for a query within the script file (the program currently does not have the capabilities to handle a JSON file query), as well as the number of K-ranked results that the user wants. If the query is correct, a local processor will open on the desired port (default = 5000), and it will show the top K-ranked results for the documents that are the most relevant to the user query.
10) If the query is incorrect, it will produce an error message, and ask the user to redo the query.

Conclusion

Objectives accomplished:

I, The program is able to successfully crawl the given initial URL, and its following pages, given a user-specified depth and page limit.

II, The system is able to clean the data retrieved from the HTML files, get the tf-ifd values and cosine similarity of the terms, and store them into an inverted index.

III, The system successfully outputs the correct results (documents) in order of relevance to the user query on a Flask-based local processor. It also handles incorrect queries and asks for a retest if the user query does not match any terms in the document collection.

Failures:

I, There are some cases where the spider either does not function correctly, breaks, or runs infinitely and has to be manually stopped.

II, The Flask processor does not take in queries as a JSON file, but rather the user has to enter the query within the actual script. This limits the variety of queries that can be entered and tested with the system.

III, The processor's local website is not very appealing and uses a very simple HTML template to output the results.

Data Sources

For the purposes of testing the crawler, the www.nirsoft.net website was used as it allows any kind of web scraping to be done on it.

Source Code

Dependencies:

```
import scrapy
from scrapy.linkextractors import LinkExtractor
from scrapy.spiders import CrawlSpider, Rule
from scrapy.exceptions import CloseSpider
import downFiles
from downFiles.items import DownfilesItem

import codecs
import os
import pandas as pd
from pandas import read_html
import html5lib
import re
import math
import sklearn
from sklearn.feature_extraction.text import CountVectorizer
from sklearn.metrics.pairwise import cosine_similarity
from collections import defaultdict
import pickle

import flask
from flask import Flask
from flask import redirect, url_for, request
from flask import render_template
import os
import pickle
import math
```

User Code:

(All of the scripts are provided in a separate a zip file ‘Scripts and Documents’)

Bibliography

"Documentation of scikit-learn 0.21.3¶," *learn*. [Online]. Available: https://scikit-learn.org/0.21/documentation.html. [Accessed: 30-Apr-2022].

"Freeware utilities: Password recovery, system utilities, Desktop Utilities - for windows," *NirSoft*. [Online]. Available: https://www.nirsoft.net/. [Accessed: 30-Apr-2022].

"Scrapy 2.6 documentation¶," *Scrapy 2.6 documentation - Scrapy 2.6.1 documentation*, 29-Apr-2022. [Online]. Available: https://docs.scrapy.org/en/latest/. [Accessed: 30-Apr-2022].

"Welcome to flask¶," *Welcome to Flask - Flask Documentation (2.1.x)*. [Online]. Available: https://flask.palletsprojects.com/en/2.1.x/. [Accessed: 30-Apr-2022].